Maintaining Doubly-Linked List Invariants in Shape Analysis with Local Reasoning [⋆]

Sigmund Cherem and Radu Rugina

Computer Science Department
Cornell University
Ithaca, NY 14853
{siggi,rugina}@cs.cornell.edu

Abstract. This paper presents a novel shape analysis algorithm with local reasoning that is designed to analyze heap structures with structural invariants, such as doubly-linked lists. The algorithm abstracts and analyzes one single heap cell at a time. In order to maintain the structural invariants, the analysis uses a local heap abstraction that models the sub-heap consisting of one cell and its immediate neighbors. The proposed algorithm can successfully analyze standard doubly-linked list manipulations.

1 Introduction

Shape analyses are aimed at extracting heap invariants that describe the "shape" of recursive data structures [1]. For instance, heap reference count invariants allow a program analyzer to distinguish acyclic and unshared data structures, such as acyclic lists or trees, from structures with sharing or cycles. Shape information has many potential applications such as: verification of heap manipulations [2]; automatic parallelization [3]; static detection of memory leaks and other heap errors [4]; and compile-time memory management [5]. Statically computing reference count invariants is challenging because destructive heap mutations temporarily break these invariants. A shape analysis must determine that the invariants are restored as the destructive operations finish.

In recent work, we have developed a novel shape analysis framework that uses local reasoning about single heap cells [4]. In this framework, the analysis uses a local abstraction to describe the state of a single heap cell. Using the local abstraction, the analysis tracks the state of the single cell through the program, from the point where the cell is allocated, and up to the point where it becomes unreachable. The single cell is referred to as the tracked cell. As shown in [4], this approach makes it possible to build efficient intra-procedural and inter-procedural analysis algorithms. However, a shortcoming of the current formulation is that it cannot accurately compute shape information for data structures with local invariants, such as doubly-linked lists.

In this paper we present a shape analysis with local reasoning about single heap cells that is capable of identifying and maintaining information about doubly-linked list invariants. We propose a new local abstraction capable of expressing such invariants. Then, we develop an analysis algorithm that computes shape information using this

[⋆] This work was supported in part by NSF grants CCF-0541217 and CNS-0406345.

abstraction. The local abstraction for a heap cell describes the local heap around the cell, consisting of the cell itself and its immediate neighboring cells. Points-to relations between the cell and its neighbors allow the analysis to express local structural invariants. The paper shows that maintaining structural invariants for the tracked cell requires knowledge about its neighbors' reference counts. Our abstraction can also express other forms of local invariants, in particular the parent-child relationship in trees with parent pointers. However, this paper mainly focuses on studying doubly-linked lists.

When a distant cell gets closer to the tracked cell and becomes one of its neighbors (for instance, when removing the element next to the tracked cell), a local analysis has no knowledge about the reference counts of the new neighbor. To address this issue, we propose an *assume-and-check* approach: when the analysis of a single cell reaches an assumption point in the program, it assumes facts about the neighbors' reference counts; at the same time, the analysis checks the reference counts of all tracked cells at that point, to ensure that the assumptions were correct.

The rest of the paper is organized as follows. Section 2 gives the background. Section 3 shows an example and discusses the issues that the analysis must overcome. Next, Section 4 presents the local abstraction and Section 5 shows the analysis algorithm. Finally, related work is discussed in Section 6 and we conclude in Section 7.

2 Background: Local Analysis of Single Heap Cells

This section discusses the key concepts behind heap analysis with local reasoning about single heap cells. The main idea is that the analysis uses a local abstraction to model one single heap cell at a time. Hence, the analysis has only local information about the one cell, but knows nothing about the rest of the heap. In contrast, traditional shape analyses that use shape graphs [6] or 3-valued logic [7] have a global view of the heap. Recent work has explored formulations using procedure-local sub-heaps [8], or using separation logic [9, 10]. Although these approaches restrict themselves to sub-heaps, their abstractions still describe entire structures (e.g., entire lists), not single cells.

Roughly speaking, an analysis that reasons about single cells is concerned with questions of the form "if property X holds for *one* heap cell before an operation, does X hold for that cell afterwards?". In contrast, global analyses answer questions of the form "if property X holds for *all* the cells before an operation, does X hold for all cells afterwards?". A local analysis is more efficient due to the finer granularity of the abstraction. However, it is more restricted because less information is available when analyzing a single cell.

The local abstraction of a heap cell is referred to as a *configuration*. The cell described by the configuration is referred to as *the tracked cell*. Each configuration contains reference counts for the tracked cell, plus additional information for accurately maintaining these reference counts. Reference counts are expressed relative to a *region partitioning* of the program's memory (both stack and heap) into a finite set of disjoint regions, so that each configuration keeps track of one reference count per region. To ensure a finite abstraction, reference counts are bound to a fixed value k per region (and a top value is used for larger counts). Usually, $k = 2$ suffices. In this paper, we assume a type-safe Java-like language, where a simple region partitioning can be constructed

```
class DLList {
   DLList n, p;
   int data;

   DLList(int d) {
     data = d;
   }
}
```

```
void insert(DLList x, int d) {
  DLList t;
  t = x.p;
  y = new DLList(d);
  y.n = x;
  y.p = t;
  t.n = y;
  x.p = y;
}
```

Fig. 1. Doubly-linked list insertion

by using one region per variable and one region per heap field. In the rest of the paper, we refer to regions using their variable or field names. The entire heap abstraction at a program point is the finite set of possible configurations at that point. However, configurations are independent, so they can be analyzed separately. The analysis uses efficient, fine-grained worklist algorithms to process individual configurations, not entire heap abstractions (in a fashion similar to attribute-independent analyses).

For a given program, the analysis generates a configuration after each allocation site, to model a *representative cell* created at that site. Then, the analysis tracks this configuration through the program using a dataflow analysis.

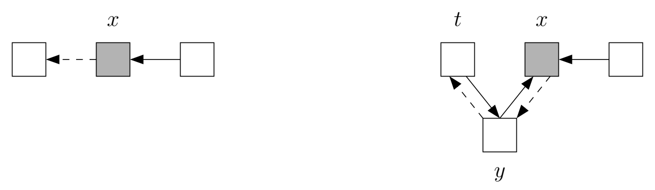
In our previous work [4, 5], each local abstraction is a triple (r, h, m), where r indicates the reference counts per region, h is a set of expressions that reference the tracked cell (or *hit* expressions); and m is a set of expressions that do not reference the cell (*miss* expressions). The h and m sets need not be complete; the richer these sets are, the more precise the analysis is. In general, redundant information is avoided, i.e., h and m exclude expressions e for which r already indicates whether e hits or misses.

For example, consider an acyclic singly-linked list, where next fields are named n. Assume that the first two list elements are pointed by variables x and y, respectively. This heap can be described using three local abstractions: $(x^1, \varnothing, \varnothing)$ describes the first list element; $(y^1n^1, \{x.n\}, \varnothing)$ describes the second list element; and $(n^1, \varnothing, \{x.n\})$ describes one list element other than the first two, that is, it describes a representative among the cells in the tail of the list. Here, reference counts are described using superscripts, and missing regions have zero reference counts by default. The analysis can analyze each of these pieces separately, reasoning locally about each of them.

However, the triples (r, h, m) cannot express local structural invariants, such as doubly-linked list invariants. In this paper we propose a new local abstraction for describing and maintaining structural invariants.

3 Example

Consider the program in Figure 1. The program is written using a Java-like syntax and is used as a running example. The program inserts a new element y in a doubly-linked list, right before element x. Each list element has a field n that points to the next element, and a field p that points to the previous element. A correct manipulation of the list must



a) Before `insert(x)` b) After `insert(x)`

Fig. 2. Counterexample: the property rc for the shaded cell holds before `insert(x)`, but not after. The shaded box denotes the tracked cell. Solid lines are next links n, and dashed lines are previous links p.

maintain the doubly-linked list (DLL) invariant:

$$\forall h \,.\, (h.p \neq \mathsf{null} \Rightarrow h.p.n = h) \wedge (h.n \neq \mathsf{null} \Rightarrow h.n.p = h)$$

3.1 Reference Counts and DLL Invariants

First, we show that maintaining precise heap reference counts requires knowledge about the DLL invariant. Consider the two predicates below for a heap cell h in a list:

- $rc(h)$ is true if it has reference counts of at most 1 from each of the fields n and p;
- $dll(h)$ indicates that the DLL invariant holds for h.

We ask the following question: given a cell h such that $rc(h)$ holds, but $dll(h)$ might not hold before insert, does $rc(h)$ hold after the insertion? The answer is negative:

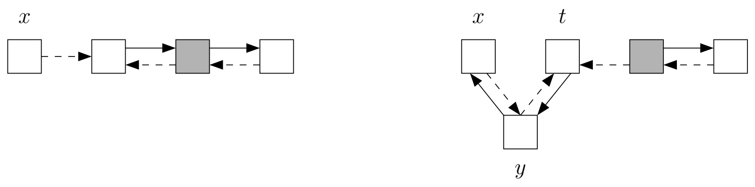
$$rc(h) \;\not\Rightarrow\; rc'(h)$$

where $rc(h)$ and $rc'(h)$ are the values of the reference counting predicate in the states before and after insert, respectively. This is shown by the counterexample in Figure 2. A concrete heap before `insert(x)` is shown on the left of the figure, and the resulting heap after the insertion is shown on the right. The cell in question h (i.e., the tracked cell) is shown using the shaded box. Next links are shown using solid lines, and previous links are shown using dashed lines. The property $rc(h)$ holds before insert, but not after, because the cell pointed to by x has two references from n fields in the result heap.

Hence, the analysis must have knowledge about the DLL invariant in order to preserve accurate reference counts during destructive doubly-linked list operations. This is the case for both local and global analyses.

3.2 Maintaining the DLL Invariant Using Local Reasoning

Next we want to determine the amount of local information needed so that a local analysis can conclude that the DLL invariant is restored. We ask the following question: if one cell h is such that both $rc(h)$ and $dll(h)$ hold before insert, is it the case that $dll(h)$



a) Before `insert(x)` b) After `insert(x)`

Fig. 3. Counterexample: a) before `insert(x)`, both the rc and dll properties hold for the shaded cell; b) after insertion, property dll doesn't hold for the shaded cell.

also holds after insert? Note that nothing is known about the rc and dll properties of elements other than h. The answer to this question is again negative:

$$rc(h) \wedge dll(h) \not\Rightarrow dll'(h)$$

This is shown by the counterexample in Figure 3. The cell in question h is the shaded cell. In the heap before insert both $rc(h)$ and $dll(h)$ hold. However, the neighboring cell to the left of h is malformed because it is referenced by two p fields, one from the tracked cell and one from x. Inserting a new element before x "steals" a reference from h and breaks its DLL property: after insertion, $h.p.n \neq h$.

Still, it is possible to determine that insert maintains the DLL invariant using local reasoning. The required piece of information is that the neighbors $h.n$ and $h.p$ of the tracked cell h also satisfy the reference count property rc before insertion [1]. The analysis can then prove that if the tracked cell satisfies rc and dll before insert, and its neighbors satisfy rc, then rc and dll hold for the tracked cell after insert:

$$rc(h) \wedge dll(h) \wedge rc(h.p) \wedge rc(h.n) \;\Rightarrow\; rc'(h) \wedge dll'(h)$$

The goal of our analysis is to build an appropriate local abstraction and prove this property using that abstraction.

4 The Local Abstraction

Based on the above observations, the local abstraction must capture: a) local invariants, such as the dll property, and b) reference counts for both the tracked cell and its neighbors. We build the abstraction as follows. The configuration of the tracked cell models the local heap consisting of itself and its immediate neighbors, i.e., those cells that are pointed by, or point to the tracked cell. The configuration models the following:

– Points-to relations between the tracked cell and its neighbors;
– Precise reference counts for the tracked cell, from each variable and each field; and
– Partial reference counts for the neighbors, from some variables and fields.

Graphically, a configuration can be thought as being a "circle" whose center is the tracked cell, and whose heap neighbors at distance 1 lie on this circle.

[1] A slightly weaker condition is actually sufficient: that $h.n$ has only one n reference, and $h.p$ has only one p reference, each of them from h.

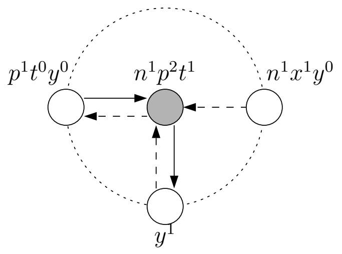


Fig. 4. Example local abstraction.

For instance, the local abstraction shown in Figure 4 arises during the analysis of `insert`. The tracked cell is the shaded node in the center. The points-to relations between the center node and its neighbors allow the analysis to express the local structural invariants. The reference counts from variables (x, y, or t) and fields (n or p) are shown using superscripts, for each node. Reference counts from variables can only be 0 or 1. For the tracked cell, the reference counts not shown (from x and y) are zero, by default. For the neighboring cells, the missing reference counts are unknown by default. Hence, reference counts are fully known for the tracked cell, but partially known for the neighbors. To explain the examples in this paper, we will refer to each local abstraction using the reference counts of the tracked cell. For instance, the above abstraction is $n^1p^2t^1$.

Note that the local abstraction does not contain summary nodes. In particular, nothing is known about the heap beyond the circle. This is the key aspect that distinguishes it from traditional global abstractions such as shape graphs.

4.1 Analysis of the Example

Figure 5 shows the analysis result for `insert` using this local abstraction. The possible local abstractions are shown at each point. In each abstraction, the tracked cell is shown as the shaded node. For simplicity, we consider only two input configurations at the entry of the function, n^1p^1 and $n^1p^1x^1$. The former describes a list cell that is not referenced by x; the latter is the cell that x references. Both cases assume that the cell in question is in the middle of the list. Four other configurations describe cases where the tracked cell is the first or the last element: n^1, n^1x^1, p^1, and p^1x^1. The analysis of those cases are similar and we omit them.

Consider the initial abstraction n^1p^1 and the first assignment `t = x.p`. The analysis tries to determine whether `x.p` is the tracked cell. Since there is not enough information to figure this out, the analysis bifurcates into two possible cases. These correspond to the first two columns in the figure. In the first case, `x.p` is not the tracked cell, so `t` will not reference the cell after the assignment. The resulting abstraction is n^1p^1. In the second case, `x.p` is the tracked cell, so t will reference it after the assignment. The resulting abstraction is $n^1p^1t^1$.

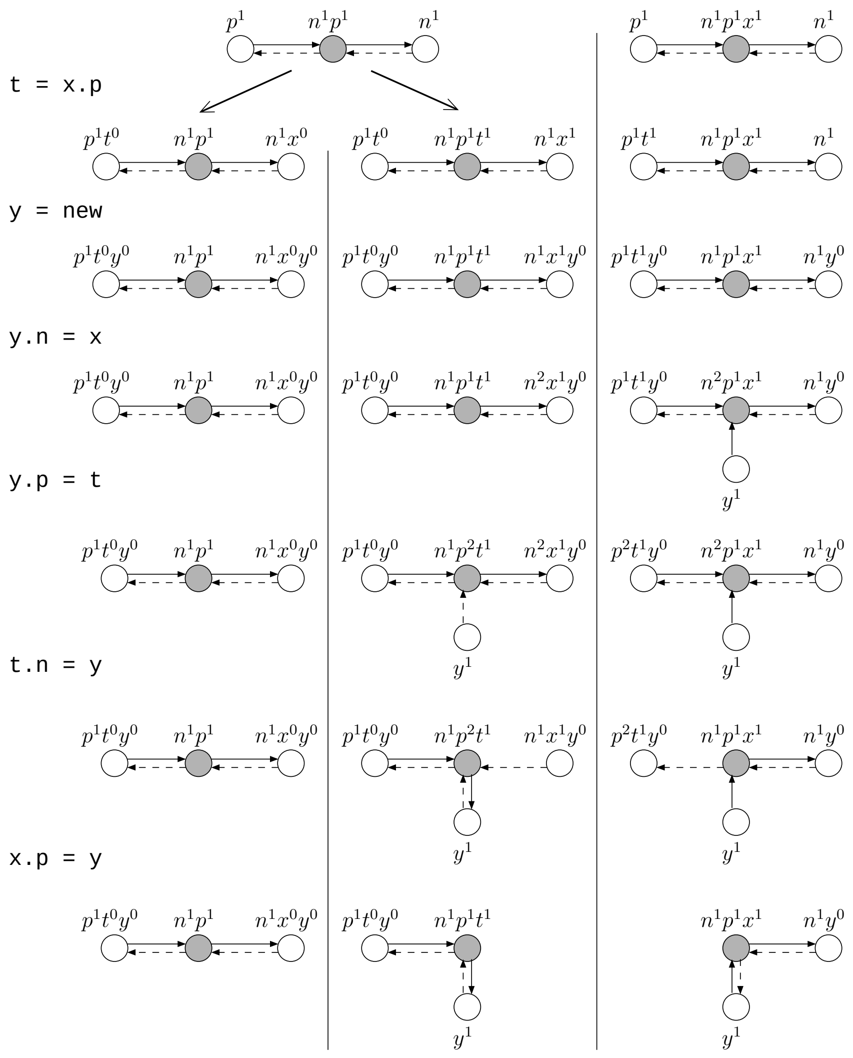

p^1 n^1p^1 n^1
p^1 $n^1p^1x^1$ n^1
t = x.p
p^1t^0 n^1p^1 n^1x^0
p^1t^0 $n^1p^1t^1$ n^1x^1
p^1t^1 $n^1p^1x^1$ n^1
y = new
$p^1t^0y^0$ n^1p^1 $n^1x^0y^0$
$p^1t^0y^0$ $n^1p^1t^1$ $n^1x^1y^0$
$p^1t^1y^0$ $n^1p^1x^1$ n^1y^0
y.n = x
$p^1t^0y^0$ n^1p^1 $n^1x^0y^0$
$p^1t^0y^0$ $n^1p^1t^1$ $n^2x^1y^0$
$p^1t^1y^0$ $n^2p^1x^1$ n^1y^0
y^1
y.p = t
$p^1t^0y^0$ n^1p^1 $n^1x^0y^0$
$p^1t^0y^0$ $n^1p^2t^1$ $n^2x^1y^0$
$p^2t^1y^0$ $n^2p^1x^1$ n^1y^0
y^1
y^1
t.n = y
$p^1t^0y^0$ n^1p^1 $n^1x^0y^0$
$p^1t^0y^0$ $n^1p^2t^1$ $n^1x^1y^0$
$p^2t^1y^0$ $n^1p^1x^1$ n^1y^0
y^1
y^1
x.p = y
$p^1t^0y^0$ n^1p^1 $n^1x^0y^0$
$p^1t^0y^0$ $n^1p^1t^1$
$n^1p^1x^1$ n^1y^0
y^1
y^1

Fig. 5. Analysis of the example program.

The analysis of `t = x.p` also infers that `x` does not reference the right neighbor in the first case (otherwise, `x.p` references the tracked cell); and that `x` references the right neighbor in the second case (because the only other cell that has a `p` field pointing to the tracked cell is the right neighbor). This information about `x` is needed later, when analyzing the assignment `x.p = y`.

Furthermore, in both cases the analysis of `t = x.p` infers that `t` doesn't reference the left neighbor, as shown by the reference count t^0. This is because the left neighbor has exactly one `p` reference, from the tracked cell. If `t` would point to the left neighbor, then `x` would reference the tracked cell, which is known to be false. Hence, the p^1

$$\begin{aligned}
&\mathit{mayAlias}(S, v_i, v_j) \Leftrightarrow (v_i \neq v_o \neq v_j \ \wedge \\
&\qquad\qquad (\forall r \,.\, ||v_i||_r = ||v_j||_r \ \vee\ ||v_i||_r = \top \ \vee\ ||v_j||_r = \top))
\end{aligned}$$

$$\mathit{hit}(S, e, v) \Leftrightarrow \begin{cases} e = x \quad \wedge\ ||v||_x = 1 \ \text{ or} \\ e = x.f \ \wedge\ ||v||_f \neq 0 \ \wedge\ (\exists v' \,.\, \mathit{hit}(S, x, v') \ \wedge\ v' \rightarrow_f v) \ \text{ or} \\ e = \mathsf{null} \wedge v = v_{\mathsf{null}} \end{cases}$$

$$\mathit{contains}(S, e) \Leftrightarrow (\exists v \in V \,.\, \mathit{hit}(S, e, v))$$

$$\mathit{miss}(S, e, v) \Leftrightarrow \begin{cases} e = x \quad \wedge\ ||v||_x = 0 \ \text{ or} \\ e = x.f \ \wedge\ ||v||_f = 0 \ \text{ or} \\ e = x \quad \wedge\ (\exists v' \,.\, ||v'||_x = 1 \ \wedge\ \neg \mathit{mayAlias}(S, v, v')) \ \text{ or} \\ e = x.f \ \wedge\ ||v||_f = 1 \ \wedge\ (\exists v' \,.\, v' \rightarrow_f v \ \wedge\ \mathit{miss}(S, x, v')) \ \text{ or} \\ e = \mathsf{null} \wedge v \neq v_{\mathsf{null}} \ \text{ or} \\ e = x.f \ \wedge \mathit{hit}(S, x, v_o) \ \wedge\ v_o \rightarrow_f v_{\mathsf{null}} \ \wedge\ \neg \mathit{mayAlias}(S, v, v_{\mathsf{null}}) \end{cases}$$

Fig. 6. Queries on configurations

knowledge for the left neighbor allows the analysis to infer that t doesn't reference that neighbor. As a result, situations such as the one in Figure 3 are not possible.

The analysis of the other statements and local abstractions is similar. The configurations at the end of the function indicate that the rc and dll properties hold for all heap cells at that point.

Abstraction Model The local abstraction is modeled as a star graph S:

$$S = (V, v_o, v_{\mathsf{null}}, O, I, ||\cdot||) \qquad \text{where,}$$

$$v_o, v_{\mathsf{null}} \in V \qquad O \subseteq \mathsf{Field} \times V \qquad I \subseteq V \times \mathsf{Field} \qquad ||\cdot|| : V \rightarrow (\mathsf{Field} \cup \mathsf{Var}) \rightarrow \mathbb{N}_\top$$

The set V contains all nodes in the graph, where $v_o \in V$ is a distinguished center node representing the tracked cell. The node $v_{\mathsf{null}} \in V$ is a special node to represent null values. The set O contains outgoing edges from v_o. A pair $(f, v) \in O$ denotes the edge $v_o \rightarrow_f v$. The special edge $v_o \rightarrow_f v_{\mathsf{null}}$ indicates that the field f of the tracked cell is null. Similarly, an incoming edge $v \rightarrow_f v_o$ is denoted by a pair $(v, f) \in I$. The cardinality function $||\cdot||$ models the reference counts for each node in V, both from variables (Var) and fields (Field). The set $\mathbb{N}_\top$ extends natural numbers with a special top value $\top$, such that $\top + 1 = \top - 1 = \top$. The heap reference count from a field f is denoted $||v||_f$. The reference counts from a variable x is denoted as $||v||_x$. If this value is not $\top$, it can only be 1 or 0, indicating whether the cell v is referenced by variable x or not. The special value $\top$ represents unknown information. As mentioned in Section 2, we use an upper bound k (e.g., $k = 2$) for the number of reference counts per field. In addition, the analysis uses a top configuration $S_\top$ to model cases where the analysis has lost precision about the tracked cell.

$$
\begin{array}{ll}
V = \{v_o, v_{\mathsf{null}}\} \cup \mathrm{range}(O) \cup \{v \mid v \in \mathrm{dom}(I) \ \wedge \ \exists x \ . \ ||v||_x = 1\} & (1) \\
v_{\mathsf{null}} \notin \mathrm{dom}(I) & (2) \\
\forall r \in \mathsf{Var} \cup \mathsf{Field} \ . \ ||v_o||_r \neq \top & (3) \\
\forall v \in V, f \in \mathsf{Field} \ . \ |\{v \rightarrow_f v' \mid v' \in V\}| = 1 & (4) \\
\forall v \ . \ ||v||_f = 1 \Rightarrow |\{v' \mid v' \rightarrow_f v\}| \leq 1 & (5) \\
\forall v_1, v_2, e \ . \ \mathit{hit}(S, e, v_1) \wedge \mathit{hit}(S, e, v_2) \Rightarrow v_1 = v_2 & (6)
\end{array}
$$

Fig. 7. Consistency invariants maintained by the algorithm.

Given a configuration S, the analysis can derive the queries presented Figure 6:

– *Alias information*. Two nodes are unaliased if any of their reference counts is inconsistent, i.e. they have different numeric values.
– *Hit expressions*. The function *hit*(S, e, v) indicates that expression e references the cell represented by the node v. This is defined recursively using the reference counts and points-to relations.
– *Miss expressions*. The function *miss*(S, e, v) indicates that e doesn't reference the cell represented by v.

We will use these queries to formalize the analysis algorithm in the next Section. Figure 7 presents several invariants that our analysis maintains at all times:

1. All nodes other than v_{null} must be directly connected to v_o. Moreover, a node v pointing into v_o ($v \rightarrow_f v_o$) must also be pointed by v_o ($v_o \rightarrow_g v$) or by some variable ($||v||_x = 1$). This invariant ensures that the number of nodes and edges in the graph is bounded by the number of variables and fields in the program.
2. Since v_{null} represents null values, it can't have outgoing edges.
3. All references to the tracked cell are precisely known.
4. A node can have at most one outgoing edge with the same field.
5. If a node v has a single incoming reference from some field f, a configuration can only have one node to represent this predecessor.
6. Each expression references at most one node.

5 Analysis Algorithm

We now proceed to present the dataflow algorithm that computes a heap abstraction at each program point. For each configuration that models the state of the tracked cell before a statement, the analysis computes a set of configurations that describes the possible states of the cell after the statement.

We assume a simple program representation consisting of a control-flow graph whose nodes are simple assignment. Assignments and expressions have the form:

$$
\begin{array}{rl}
\text{Statements} & s ::= x = \mathsf{new} \mid x = \mathsf{null} \mid x = y \mid x = y.f \mid x.f = y \mid x.f = \mathsf{null} \\
\text{Expressions} & e ::= \mathsf{null} \mid x \mid x.f
\end{array}
$$

where $x \in \mathsf{Var}$ ranges over variables, and $f \in \mathsf{Field}$ ranges over fields.

$$
\begin{array}{l}
\textit{focusH}(S, x.f) = (V', v_o', v_{\mathsf{null}}', O', I'', ||\cdot||') \quad \text{where,} \\
\quad S' = \begin{cases} \textit{unify}(\textit{addNode}(S, v_x, x, 1), v_x, v) & \neg\textit{contains}(S, x) \ \wedge \ ||v_o||_f = 1 \ \wedge \ v \rightarrow_f v_o \\ \textit{addNode}(S, v_x, x, 1) & \neg\textit{contains}(S, x) \quad v_x \text{ fresh} \\ \textit{unify}(S, v_x, v) & ||v_o||_f = 1 \ \wedge \ v \rightarrow_f v_o \\ S & \text{otherwise} \end{cases} \\
\quad I'' = I' \cup \{v_x \rightarrow_f v_o\}
\end{array}
$$

$$
\begin{array}{l}
\textit{focusM}(S, x.f) = (V', v_o', v_{\mathsf{null}}', O', I'', ||\cdot||') \quad \text{where,} \\
\quad S' = \begin{cases} \textit{unify}(\textit{addNode}(S, v', x, 0), v', v) & ||v_o||_f = 1 \ \wedge \ v \rightarrow_f v_o \quad v' \text{ fresh} \\ \textit{addNode}(S, v', x, 0) & ||v_o||_f = 1 \quad v' \text{ fresh} \\ S_\top & \text{otherwise} \end{cases} \\
\quad I'' = I' \cup \{v' \rightarrow_f v_o\}
\end{array}
$$

Fig. 8. Focus operations. The helper functions *addNode* and *unify* are defined in Figure 11. We use S' as a shorthand notation for $(V', v_o', v_{\mathsf{null}}', O', I', ||\cdot||')$.

Initialization As discussed in Section 2, for each allocation site $x = \mathsf{new}$, the analysis builds a configuration $S = (\{v_o, v_{\mathsf{null}}\}, v_o, v_{\mathsf{null}}, \varnothing, \{v_o \rightarrow_f v_{\mathsf{null}} \mid f \in \mathsf{Field}\}, ||\cdot||)$ at the program point after the allocation, where $||v_o||_x = 1$ and $||v_o||_r = 0$ for any $r \neq x$. The configuration describes a representative heap cell allocated at this site. Then, the analysis tracks this configuration through the program.

Alternatively, if a code fragment is to be analyzed separately, the set of all possible configurations at the beginning of that fragment must be supplied.

Focus operations Given an input configuration describing the state of the tracked cell before an assignment statement $e_1 = e_2$, the analysis tries to determine whether e_1 and e_2 reference the tracked cell. Whenever the analysis cannot determine if e_i ($i \in \{1, 2\}$) hits or misses the tracked cell (i.e. $\neg\textit{hit}(S, e_i, v_o) \wedge \neg\textit{miss}(S, e_i, v_o)$), the analysis bifurcates and creates two new configurations that are focused with respect to e_i.

Figure 8 shows the focus operations. Since exact reference counts are known for v_o, it is known whether variables hit or miss v_o. Therefore, the analysis only focuses expressions of the form $x.f$. To make an expression $x.f$ hit v_o, the analysis simply unifies the predecesor of v_o via field f (v) and the node referenced by x (v_x). The operation will also add the node v_x or the incoming field f if they didn't exist before focusing. A similar algorithm is used to make an expression $x.f$ miss v_o. Although, if $||v_o||_f \geq 2$, it is not possible to express the fact that $x.f$ misses the object. If this situation occurs, the focus operation returns an imprecise configuration $S_\top$ indicating that the analysis no longer tracks the state of the tracked cell.

Transfer function The analysis then applies the transfer function to each focused configuration. Figure 9 presents the transfer function for an assignment $e_1 = e_2$. First, the analysis nullifies e_1 using the helper function *kill*. For store assignments $x.f = y$, the analysis also creates the node for y in case it didn't exist, as this node might become a

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textit{transfer}(S, e_1 = e_2) = \textit{clean}(V', v'_o, v'_{\mathsf{null}}, O'', I'', ||\cdot||'') \quad \text{where,}} \\
S' & = & \begin{cases} \textit{addNode}(\textit{kill}(S, e_1), v', y, 1) & e_1 = x.f \ \wedge \ e_2 = y \ \wedge \ \neg\textit{contains}(S, y) \\ \textit{kill}(S, e_1) & \text{otherwise} \end{cases} \\
||v||''_r & = & \begin{cases} ||v||'_r + 1 & \textit{hit}(S, e_2, v) \quad \wedge [(e_1 = x \wedge r = x) \ \vee \ (e_1 = x.f \wedge r = f)], \ \text{ or} \\ ||v||'_r & \textit{miss}(S, e_2, v) \vee (e_1 = x \wedge r \neq x) \ \ \vee \ (r \neq f \wedge e_1 = x.f), \ \text{ or} \\ \top & \text{otherwise} \end{cases} \\
O'' & = & O' \cup \{v_o \rightarrow_f v \mid e_1 = x.f \ \wedge \ \textit{hit}(S', x, v_o) \ \wedge \ \textit{hit}(S', e_2, v)\} \\
I'' & = & I' \ \cup \{v \rightarrow_f v_o \mid e_1 = x.f \ \wedge \ \textit{hit}(S', x, v) \ \wedge \ \textit{hit}(S', e_2, v_o)\}
\end{array}
$$

Fig. 9. Transfer function. The helper functions *addNode*, *kill* and *clean* are defined in Figure 11. We use S' as a shorthand notation for $(V', v'_o, v'_{\mathsf{null}}, O', I', ||\cdot||')$.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textit{merge}(S^1, S^2) = \textit{clean}(V', v'_o, v'_{\mathsf{null}}, O', I', ||\cdot||') \quad \text{where,}} \\
V' & = & \{v_{i,j} \mid v_i \in V^1 \wedge v_j \in V^2\} \\
v'_o & = & v_{o,o} \\
v'_{\mathsf{null}} & = & v_{\mathsf{null},\mathsf{null}} \\
O' & = & \{v'_o \rightarrow_f v_{i,j} \mid v^1_o \rightarrow^1_f v_i \ \wedge \ v^2_o \rightarrow^2_f v_j\} \\
I' & = & \{v_{i,j} \rightarrow_f v'_o \mid v_i \rightarrow^1_f v^1_o \ \wedge \ v_j \rightarrow^2_f v^2_o\} \\
||v_{i,j}||'_r & = & ||v_i||^1_r \ \sqcup \ ||v_j||^2_r
\end{array}
$$

Fig. 10. Merge operation. Precondition: $||v^1_o|| = ||v^2_o||$ and $(v^1_o \rightarrow^1_f v^1_o \Leftrightarrow v^2_o \rightarrow^2_f v^2_o)$.

neighbor after the store. The reference counts are then updated. The appropriate reference count of each node v is increased when e_2 hits v, it remains unchanged when e_2 misses, and it is set to $\top$ when the analysis cannot determine whether e_2 hits or misses. The points-to edges are added in the case of store statements. Finally, the *clean* helper function removes nodes that are not neighbors of the tracked cell.

Merge operation At join points, the analysis uses the merge operation from Figure 10 to combine configurations from different branches. Two configurations are combined only if they have identical reference counts and the same set of self-edges on the tracked cell. The merge operation defines one node for each pair of nodes in the input configurations. The reference counts are combined using the join in the flat lattice $(\mathbb{N}_\top, \sqsubseteq)$. Thus, if $i \neq j$: $i \sqcup i = i$, $i \sqcup j = \top$, and $i \sqcup \top = \top \sqcup i = \top$. The *clean* operation guarantess that the number of nodes and edges in the resulting configuration is bounded by the number of variables and fields in the program.

Auxiliary functions The auxiliary operations used by the analysis are fairly straightforward. They are shown in Figure 11 and are summarized below:

– The *addNode* operation adds a neighboring node, without connecting it to v_o. The reference count of the added node from variable x is set according to $i \in \{0, 1\}$. This function is used both when focusing and when applying the transfer function.

$$
\begin{array}{lll}
\textit{addNode}(S, v', x, i) = & (V \cup \{v'\}, v_o, v_{\mathsf{null}}, O, I, ||\cdot||') \quad \text{where,} \\
& ||v||'_r = \begin{cases} i & r = x \ \wedge\ v = v' \\ \top & r \neq x \ \wedge\ v = v' \\ ||v||_r & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{lll}
\textit{kill}(S, e) & = & (V, v_o, v_{\mathsf{null}}, O - K, I - K, ||\cdot||') \text{ where,} \\
& ||v||'_r = & \begin{cases} 0 & e = r = x \\ ||v||_r - 1 & e = x.f \ \wedge\ r = f \ \wedge\ \textit{hit}(S, e, v) \\ ||v||_r & \text{otherwise} \end{cases} \\
& K & = \{v \rightarrow_f v' \mid e = x.f \ \wedge\ \neg \textit{miss}(S, x, v)\}
\end{array}
$$

$$
\begin{array}{lll}
\textit{clean}(S) & = & (V', v_o, v_{\mathsf{null}}, O, V' \lhd I, V' \lhd ||\cdot||) \text{ where,} \\
& V' & = \{v_o, v_{\mathsf{null}}\} \cup \text{range}(O) \cup \{v \mid v \in \text{dom}(I) \ \wedge\ \exists x \,.\, ||v||_x = 1\} \\
& & \text{where } V' \lhd f \text{ restricts the domain of } f \text{ to } V'
\end{array}
$$

$$
\begin{array}{lll}
\textit{unify}(S, v_i, v_j) & = & (V', v_o, v_{\mathsf{null}}, O', I', ||\cdot||') \text{ where,} \\
& V' & = V - \{v_j\} \\
& O' & = O - \{v_o \rightarrow_f v_j\} \cup \{v_o \rightarrow'_f v_i \mid v_o \rightarrow_f v_j \in O\} \\
& I' & = I \ - \{v_j \rightarrow_f v_o\} \cup \{v_i \rightarrow'_f v_o \mid v_j \rightarrow_f v_o \in I\} \\
& ||v||'_r = & \begin{cases} ||v_i||_r \sqcap ||v_j||_r & v = v_i \\ ||v||_r & \text{otherwise} \end{cases}
\end{array}
$$

Fig. 11. Helper operations. The function *unify* assumes *mayAlias*(S, v_i, v_j) holds, and $v_j \neq v_{\mathsf{null}}$.

– The *kill* operation removes an expression and updates the reference counts accordingly. The operation supports strong updates when field expressions are killed.
– The *clean* operation removes unnecessary nodes from a configuration. This operation is used by the end of the transfer functions and merge operation.
– The *unify* operation combines two nodes that may alias into one single node. This is done by transferring all information from one node to the other. Moreover, the result has the most precise reference counts from the input nodes.

5.1 Assume-And-Check Approach

Although the analysis can successfully determine that the reference count property rc and the doubly-linked list invariant are preserved for the tracked cell during destructive operations, in many cases it cannot determine that the reference count property of the neighbors is restored. For instance, in the `insert` example from Figure 5 the heap reference counts are not known for the neighboring cell pointed by y, because y "came from the outside" to join the local heap. A similar situation occurs when removing an element from a list: a cell two levels of indirection away from the tracked cell gets closer and becomes one of its neighbors. As discussed, the neighbor's reference count information is, however, needed before `insert`.

We address this issue using an assume-and-check approach. This approach is based on defining *assumption points* in the program. We consider that such points are manually marked by the user using a special `assume-and-check` instruction. The assumption points are program points where the analysis can safely restore the reference count information for the neighbors. As implied by the name, the analysis performs two tasks when it reaches such points:

- **Assume:** Whenever the analysis of a tracked cell reaches an assumption point, it assumes that the reference count property rc holds for all of its neighbors. More precisely, all neighbors are assumed to have at most one reference from each field. This enables the analysis to restore their reference counts: if the current configuration is such that the tracked cell points to neighbor v via some field f, i.e., $v_o \rightarrow_f v$, then the analysis restores v's reference count from f: $||v||_f = 1$.
- **Check:** Whenever the analysis of a tracked cell reaches an assumption point, it checks if the tracked cell itself satisfies the reference count property rc, i.e., if it has at most one reference per field. When the assumption is violated, the analysis reports an error and all of the analysis results are invalidated. Otherwise, if all checks succeed, then all assumptions were correct.

Essentially, restoring the reference counts of the neighbors requires knowledge about all cells. The assume-and-check approach provides a simple mechanism for gathering such global information without breaking the local analysis methodology.

Standard heap operations typically require one single assumption point, after the operation finishes. In the example from Section 3, an assume-and-check instruction is added at the end of the function. This suggests that default assumption at such points could be used to reduce the amount of annotations. In addition, assume-and-check instructions can be refined to indicate the specific field for which the reference count must be assumed and checked.

The assumptions presented here are specifically formulated for doubly-linked lists. Other shapes might require different assumptions. For instance, in the case of trees with parent pointers, the analysis must assume and check that the sum of the reference counts from `left` and `right` fields is at most one, i.e., no cell is pointed by both a `left` and a `right` link.

5.2 Soundness

This section summarizes the formal framework and the soundness result for our analysis. We refer the reader to a technical report [11] for a detailed presentation of the formal model and the complete proofs.

Each concrete program state $\sigma = (\varphi, h) \in \mathsf{State}$ consists of a variable environment φ that maps variables to values, and a heap h that maps the fields of each location to their values. Values are either heap locations (Loc) or the constant null. By abuse of notation, we write $l \in \sigma$ to indicate that l is an allocated heap cell, i.e. in the range or domain of φ or h. The execution of the program is modeled using denotational semantics via a function $[\![s]\!] : \mathsf{State} \rightarrow \mathsf{State}$ that maps the state σ before a statement s, to the state $[\![s]\!](\sigma)$ after the statement. An abstraction function $\alpha_\sigma(l)$ maps each heap cell l in a concrete heap to a local abstraction $S = (V, v_o, v_{\mathsf{null}}, O, I, ||\cdot||)$. The relation $\sqsubseteq$ is the partial order over local abstractions. An entire heap abstraction A consists of a finite set of local abstractions S. The main result is as follows.

Program	Local Abs.				Global Abs. (TVLA)		
	Configs.		Avg. Nodes	Time	Avg.	Avg. Nodes	Time
	In	Avg.	per Config.	(sec)	Structures	per Struct.	(sec)
insertBefore	7	6.5	2.2	0.07	2.7	3.9	0.59
appendLast	4	4.5	2.1	0.06	4.6	3.7	0.77
concat	4	4.5	2.3	0.07	4.8	3.7	0.88
copy	4	4.5	2.1	0.09	4.8	3.5	1.24
insertNth	4	6.2	2.3	0.09	7.0	3.2	1.38
removeData	3	8.2	2.3	0.13	10.1	3.0	1.86
filter	3	26.3	2.0	0.37	24.7	2.2	4.19

Table 1. Analysis Evaluation

Theorem 1. *Given a program P, program point p, a concrete state σ that can arise at point p during the execution of the program, and an abstraction A that the analysis computes at that program point, then each concrete heap cell in σ is modeled by at least one local abstraction in A: $\forall l \in \sigma \,.\, \exists S \in A \,.\, \alpha_\sigma(l) \sqsubseteq S$.*

The correctness proof is divided into four lemmas regarding the correctness of each of the following: the generating function at allocation sites; the transfer functions; the focus operation; and the assume-and-check coercions. The correctness of transfer functions forms the bulk of the proof.

5.3 Evaluation

We have developed a prototype implementation of the local analysis presented in this paper in Java, and used it to analyze the doubly-linked list programs shown in Table 1. Our local analysis has successfully verified that all of these programs maintain the doubly-linked list shape. All of the experiments were run on a 2GHz Pentium machine with 1GB of memory, running Linux.

The input to each program is described using at least 3 configurations (one for the middle, and one for each end of the list). Additional configurations are needed to indicate where the arguments point in the list. Programs that allocate new heap cells also include one configuration for the allocation site. The number of input configurations is shown in the first column of the table.

Each program, except `filter`, has been annotated with one single assume-and-check instruction, inserted at the end of the program. The `filter` program uses a loop to remove several elements from the input list. For this program, and additional assume annotation has been added at the beginning of the loop body. This ensures that the rc property holds on the neighboring cells after every removal from the list. The analysis successfully verifies the checks at all of the assumption points.

The data in Table 1 shows several statistics about our analysis: the average number of configurations per program point; the average number of nodes per configuration (excluding the null node); and the analysis running time. These results show that the analysis is fast, with an average running time of about 0.1 seconds per program.

To compare our implementation to a global analysis, we have also tested an implementation in TVLA [12]. We have added an instrumentation predicate to describe the DLL invariant. However, no global predicates, such as reachability, were included in this implementation. The right part of Table 1 shows the results obtained with TVLA. We observe that the number of 3-valued structures per program point is roughly equal to the number of configurations per program point in our analysis, but the number of nodes in those structures is larger than the number of nodes per configuration. Furthermore, the running time of the TVLA implementation is about 10 times slower. We attribute this in part to the fact that TVLA uses of a global abstraction, and in part to the fact that the TVLA engine is generic, while ours is specialized.

6 Related Work

The work on shape analysis dates back to Jones and Muchnick [13]. They developed a dataflow analysis for identifying (the lack of) cyclicity and sharing in heap structures using k-limited abstract heaps. Since then, many different approaches based on dataflow analysis and abstract interpretation have been proposed to address this problem [14–18, 6, 7, 19, 20, 8, 9]. Existing techniques include analyses that use path matrices and or matrices that describe reachability [15, 18], reference counting analyses [14], analyses that use shape graphs [21, 19, 6], shape analyses and abstractions expressed using three-valued logic [22, 23, 7, 8]. In addition, heap verification techniques using model-checking or Hoare logic has also been explored [24–26]. Unlike abstract interpretation, logic-based tools rely on theorem provers and typically require heavyweight loop annotations. Alternatively, it is possible to synthesize loop invariants via predicate abstraction [26–30]. The common aspect of all of the above techniques is that the analyzer or verifier requires a global view of the entire heap in order to analyze a particular piece of computation. In contrast, the analysis in this paper and our earlier analysis [4] are fundamentally different, as the analysis has knowledge about the local properties of one single heap cell, but is oblivious to the way the rest of the heap is structured. This fine-grained abstraction leads to efficient algorithms. This is achieved at the expense of giving up on global properties (such as reachability) that involve reasoning about unbounded sets of cells.

This paper follows our initial work on shape analysis with tracked heap cells [4]. The contribution of this work is a new local heap abstraction that expresses local structural invariants, and the development of an analysis that uses this abstraction to maintain these invariants. This algorithm makes shape analysis with local reasoning about single cells applicable to an important class of heap structures.

A related direction of research is the recent work on separation logic [31, 32]. This line of research has explored extensions of Hoare logic for reasoning about mutable heap structures, by providing features such as the separating conjunction and the frame rule, that makes it easier to write correctness proof for heap-manipulating programs. Recently, separation logic has also been applied to the shape analysis problem [10, 9]. Although the state transformers modify local portions of the abstract heap, their abstractions still describe entire linked structures. For instance, operations such as inserting or removing elements from a list require knowing that the entire list is well-formed, using a “listness” predicate ls. This predicate behaves similarly to the summary node in standard shape analyses; it describes a global invariant for the entire list, not a local property of a single cell.

7 Conclusions

We have presented an abstraction and analysis algorithm that makes it possible to apply shape analysis with local reasoning to data structures that maintain structural invariants, such as doubly-linked lists. The local abstraction of a cell describes the local heap around that cell, and is therefore able to express local structural invariants. The algorithm can successfully show that standard operations such as doubly-linked list insertions or removals maintain the doubly-linked list invariant.

References

1. Wilhelm, R., Sagiv, M., Reps, T.: Shape analysis. In: Proceedings of the 2000 International Conference on Compiler Construction, Berlin, Germany (2000)
2. Lev-ami, T., Reps, T., Sagiv, M., Wilhelm, R.: Putting static analysis to work for verification: A case study. In: Proceedings of the 2000 International Symposium on Software Testing and Analysis. (2000)
3. Ghiya, R., Hendren, L., Zhu, Y.: Detecting parallelism in C programs with recursive data structures. In: Proceedings of the 1998 International Conference on Compiler Construction, Lisbon, Portugal (1998)
4. Hackett, B., Rugina, R.: Region-based shape analysis with tracked locations. In: Proceedings of the 32th Annual ACM Symposium on the Principles of Programming Languages, Long Beach, CA (2005)
5. Cherem, S., Rugina, R.: Compile-time deallocation of individual objects. In: Proceedings of the International Symposium on Memory Management, Ottawa, Canada (2006)
6. Sagiv, M., Reps, T., Wilhelm, R.: Solving shape-analysis problems in languages with destructive updating. ACM Transactions on Programming Languages and Systems **20**(1) (1998) 1–50
7. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems **24**(3) (2002)
8. Rinetzky, N., Sagiv, M., Yahav, E.: Interprocedural shape analysis for cutpoint-free programs. In: Proceedings of the 12th International Static Analysis Symposium, London, UK (2005)
9. Distefano, D., O’Hearn, P., Yang, H.: A local shape analysis based on separation logic. In: Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Vienna, Austria (2006)
10. Gotsman, A., Berdine, J., Cook, B.: Interprocedural shape analysis with separated heap abstractions. In: The 13th International Static Analysis Symposium, Seoul, Korea (2006)
11. Cherem, S., Rugina, R.: Maintaining structural invariants in shape analysis with local reasoning. TR CS TR2006-2048, Cornell University (2006)
12. Lev-Ami, T., Sagiv, M.: TVLA: A system for implementing static analyses. In: Proceedings of the 7th International Static Analysis Symposium, Santa Barbara, CA (2000)
13. Jones, N., Muchnick, S.: Flow analysis and optimization of Lisp-like structures. In: Conference Record of the 6th Annual ACM Symposium on the Principles of Programming Languages, San Antonio, TX (1979)

14. Chase, D., Wegman, M., Zadek, F.: Analysis of pointers and structures. In: Proceedings of the SIGPLAN '91 Conference on Program Language Design and Implementation, White Plains, NY (1990)
15. Hendren, L., Nicolau, A.: Parallelizing programs with recursive data structures. IEEE Transactions on Parallel and Distributed Systems **1**(1) (1990) 35–47
16. Hendren, L., Hummel, J., Nicolau, A.: A general data dependence test for dynamic, pointer-based data structures. In: Proceedings of the SIGPLAN '94 Conference on Program Language Design and Implementation, Orlando, FL (1994)
17. Deutsch, A.: Interprocedural may-alias analysis for pointers: Beyond k-limiting. In: Proceedings of the SIGPLAN '94 Conference on Program Language Design and Implementation, Orlando, FL (1994)
18. Ghiya, R., Hendren, L.: Is is a tree, a DAG or a cyclic graph? A shape analysis for heap-directed pointers in C. In: Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages, St. Petersburg Beach, FL (1996)
19. Chong, S., Rugina, R.: Static analysis of accessed regions in recursive data structures. In: Proceedings of the 10th International Static Analysis Symposium, San Diego, CA (2003)
20. Rugina, R.: Quantitative shape analysis. In: Proceedings of the 11th International Static Analysis Symposium, Verona, Italy (2004)
21. Sagiv, M., Reps, T., Wilhelm, R.: Solving shape-analysis problems in languages with destructive updating. In: Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages, St. Petersburg Beach, FL (1996)
22. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: Proceedings of the 26th Annual ACM Symposium on the Principles of Programming Languages, San Antonio, TX (1999)
23. Rinetzky, N., Sagiv, M.: Interprocedural shape analysis for recursive programs. In: Proceedings of the 2001 International Conference on Compiler Construction, Genova, Italy (2001)
24. Moller, A., Schwartzbach, M.: The pointer assertion logic engine. In: Proceedings of the SIGPLAN '01 Conference on Program Language Design and Implementation, Snowbird, UT (2001)
25. McPeak, S., Necula, G.: Data structure specification via local equality axioms. In: Proceedings of the 2005 Conference on Computer-Aided Verification, Seattle, WA (2005)
26. Lahiri, S., Qadeer, S.: Verifying properties of well-founded linked lists. In: Proceedings of the 33th Annual ACM Symposium on the Principles of Programming Languages, Charleston, SC (2006)
27. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.: Automatic predicate abstraction of C programs. In: Proceedings of the SIGPLAN '01 Conference on Program Language Design and Implementation, Snowbird, UT (2001)
28. Balaban, I., Pnueli, A., Zuck, L.D.: Shape analysis by predicate abstraction. In Cousot, R., ed.: VMCAI. Volume 3385 of Lecture Notes in Computer Science., Springer (2005) 164–180
29. Dams, D., Namjoshi, K.S.: Shape analysis through predicate abstraction and model checking. In Zuck, L.D., Attie, P.C., Cortesi, A., Mukhopadhyay, S., eds.: VMCAI. Volume 2575 of Lecture Notes in Computer Science., Springer (2003) 310–324
30. Bingham, J.D., Rakamaric, Z.: A logic and decision procedure for predicate abstraction of heap-manipulating programs. In Emerson, E.A., Namjoshi, K.S., eds.: VMCAI. Volume 3855 of Lecture Notes in Computer Science., Springer (2006) 207–221
31. Reynolds, J.: Separation logic: A logic for shared mutable data structures. In: Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science, Copenhagen, Denmark (2002)
32. Ishtiaq, S., O'Hearn, P.: BI as an assertion language for mutable data structures. In: Proceedings of the 28th Annual ACM Symposium on the Principles of Programming Languages, London, UK (2001)